Diseño de Compiladores

Intérprete de Ruby

5 de Julio 2012

Casiraghi Martín, Marcelo 4.703.481-6 (marcelocasiraghi@gmail.com)
Rodríguez Blanco, Martín 4.123.589-6 (merodriguezblanco@gmail.com)
Suttner Michelena, Sebastian 4.216.541-8 (sebastian.suttner@gmail.com)

Contenido

Introducción

Nos planteamos como objetivo construir un intérprete de Ruby, utilizando las herramientas Flex y Bison estudiadas en el curso. Buscamos lograr una interpretación, ejecución y comportamiento similar al que provee el intérprete por defecto del lenguaje Ruby.

Si bien logramos el objetivo (con algunas consideraciones que aclararemos a continuación), conseguimos bajar a tierra lo estudiado y aprendido en el curso.

A lo largo del documento explicaremos las decisiones tomadas, problemas encontrados y sus soluciones. Hemos estructurado los temas, siguiendo el hilo conductor propuesto por el curso. Comenzamos explicando cómo manejamos el Frontend, para luego describir detalles de cómo encaramos el Backend. Finalizaremos con aclaraciones sobre comportamientos en concreto de nuestro intérprete.

Herramientas Utilizadas

Decidimos utilizar los utilitarios Flex y Bison para realizar análisis léxico y sintáctico respectivamente. A nivel de codificación extra, para crear análisis semántico, estructuras y ambiente de ejecución optamos por utilizar el lenguaje de programación C.

Como se vio en el curso, Flex es una herramienta desarrollada en el lenguaje C, cuya tarea es la generación de un scanner también escrito en C. El scanner se especifica utilizando expresiones regulares y código C para las acciones ejecutadas cuando una expresión regular es conocida.

Bison es un generador de análisis sintáctico de propósito general que convierte una descripción gramatical para una gramática libre de contexto LALR(1) en un programa C que analice esa gramática.

Para el control de versiones del proyecto, utilizamos el software Git diseñado por Linus Torwalds y creamos un repositorio remoto privado en GitHub para su almacenamiento. Esto nos permitió una gran flexibilidad al momento de trabajar por separado en distintas partes del proyecto, evitando que cada uno de los compañeros pisara el trabajo del otro.

El obligatorio lo desarrollamos y testeamos en máquinas con sistema operativo Ubuntu 12.04 y MacOs sobre Unix, y utilizamos la versión 4.6.3 de gcc para compilar las librerías de C del proyecto. Optamos por desarrollar el programa en el lenguaje C en lugar de C++ debido a su flexibilidad para el casteo de tipos. En contrapartida, no contamos con la posibilidad de crear clases y aprovechar la herencia.

Análisis Léxico

Buscamos definir los TOKENs que debemos reconocer, así como las expresiones regulares que los representan y así, mediante el Flex, separar la representación física en una secuencia de elementos. Luego, la cadena de TOKENs reconocidos se utiliza en el parser para construir la estructura del programa.

Para llevar a cabo el análisis léxico creamos el archivo flex ruby_lex_analyzer.l que contiene el código Flex para reconocer los TOKENs. La estructura del mismo es la siguiente:

```
%{
   Declarations
%}
   Definitions
%%
   Rules
%%
   User subroutines
```

A su vez, creamos el archivo bison ruby_grammar.y que provee al escanner una tabla con los tokens del lenguaje (ruby_grammar.tab.h) generada al compilar el archivo bison. Cabe destacar que, tanto para el análisis léxico como para la construcción el árbol sintáctico para el análisis sintáctico, nos guiamos fuertemente por la implementación que brinda el libro [...] de la bibliografía.

En la sección del lexer destinada a las reglas, declaramos cada una de las expresiones regulares necesarias para reconocer los tokens de la gramática. Creemos que vale la pena destacar aquellas expresiones que escanean tokens con comportamiento complejo. Tal es el caso de:

- Comentarios de una y de múltiples líneas.
- Inclusión de archivos ruby mediante “require” y “load”.
- Identificadores y cadenas de caracteres.
- Acceso a elementos de un arreglo.

Comentarios

Para implementar los comentarios de una sola línea declaramos en la sección de reglas la expresión regular #.*. En el caso de los comentarios multilínea utilizamos estados.

Los estados son una funcionalidad provista por Flex que permiten controlar los patrones que queremos machear y el momento en que queremos hacerlo. Para utilizarlos, incluimos en la sección de definiciones la línea %x COMMENT, que define un estado inicial que utilizamos para buscar los comentarios.

La línea %x COMMENT significa que COMMENT es un estado inicial exclusivo que implica que, cuando este estado se encuentra activo, solamente se pueden machear patrones especificados por el mismo. En el código de acción de la expresión regular, cuando se termina de machear el comentario, utilizamos la macro BEGIN para cambiar al estado INITIAL (estado inicial por defecto en Flex). A continuación copiamos el código a modo de ejemplificación:

```
%x COMMENT
...
%%
...
^=begin[ \n]       { BEGIN COMMENT;              }
<COMMENT>^=end     { BEGIN INITIAL;              }
<COMMENT>.|\n      { ;                           }
<COMMENT><<EOF>>   { return yyerror("comment"); }
...
```

Como se puede apreciar, en caso de encontrarse el caracter de fin de archivo EOF, se debe retornar un error al parser ya que no se marco en el archivo ruby el fin de comentario.

Inclusión de archivos

Para implementar la inclusión de archivos de ruby nos basamos en las estructuras y algoritmos presentados en el capítulo 2 de [...]. Explicamos brevemente la implementación de los patrones y estructuras utilizadas, una explicación mucho más detallada se encuentra en la bibliografía mencionada.

A continuación explicamos como implementamos el macheo y la implementación del método “load”. La implementación del método “require” es análoga, siendo la única diferencia que el archivo especificado se incluye una única vez. Por esta razón la obviaremos.

En primer lugar, definimos al igual que para los comentarios multilínea un estado inicial denominado LOAD_INCLUSION en la sección de definiciones del lexer. Además, definimos en la sección de declaraciones del lexer la estructura “buffer_stack” que mantiene una entrada en la lista de archivos de entrada guardados.

Luego, declaramos las siguientes expresiones regulares con su código de activación correspondiente:

```
^load[ \t]*[\"]                  { BEGIN LOAD_INCLUSION; }
<LOAD_INCLUSION>[^\"]+           {
                                    {
                                      int c;
                                      while ((c = input()) && c != '\n') {};
                                    };
                                    yylineno++;
                                    if (!open_file(strdup(yytext))) {
                                      yyterminate();         /* No such file. */
                                    }
                                    BEGIN INITIAL;
                                  }
<LOAD_INCLUSION>.|\n              {
                                    file_bad_inclusion_line_error(yylineno);
                                    yyterminate();
                                  }
<<EOF>>                           {
                                    if (!close_file()) {
                                      yyterminate();
                                    }
                                  }
```

La primer expresión regular machea la palabra “load” y la primer doble comilla previa al nombre del archivo a ser incluido. Al finalizar el patrón se pasa al estado LOAD_INCLUSION. En este estado declaramos un patrón que permite obtener el nombre del archivo a ser incluido junto con la segunda doble comilla. El nombre del archivo es pasado a la función “open_file” que se encarga de guardar el archivo actual en el buffer_stack y preparar el siguiente nivel de entrada. Dicha función se encuentra definida en la sección de subrutinas de usuario del lexer.

Pero previo a la llamada “open_file” se incluyen dentro del estado LOAD_INCLUSION dos patrones más que se encargan del resto de lo que se encuentre luego de machear el nombre del archivo, en cuyo caso se retorna el error que corresponda.

En caso de encontrarse el caracter EOF de fin de archivo, se llama a la función “close_file” definida en la sección de subrutinas de usuario del lexer. Ésta permite retornar al archivo de entrada previo al incluido. Si “close_file” retorna cero, significa que el archivo escaneado era el último abierto, por lo que se se termina la ejecución, en otro caso se continua con el escaneo del archivo de entrada previo.

La función “open_file” mencionada anteriormente prepara para leer del archivo a ser incluido, salvando los archivos de entrada previos. Para ello mantiene una estructura de lista encadenada con elementos de tipo “buffer_stack” que almacenan un link al buffer_stack previo junto con el yylineno y el nombre del archivo. La rutina “close_file” realiza exactamente lo inverso a la rutina “open_file”. Esto es, cierra el archivo abierto actualmente, borra el buffer actual y hace un restore del previo.

Identificadores y cadenas de caracteres

Tomamos como decisión de diseño representar como identificadores a las variables (tanto comunes como de instancia) y nombres de funciones. Para esto definimos el siguiente patrón junto con su código de acción:

```
[@]?[a-zA-Z_@][a-zA-Z0-9_]*(\?)? {
                                   yylval.string = malloc((strlen(yytext)+1)*sizeof(char));
                                   strcpy(yylval.string, yytext);
                                   return IDENTIFIER;
                                 }
```

Como se puede apreciar, se almacena en yylval la cadena de caracteres escaneada (que se almacena en yytext) y retornamos el token IDENTIFIER. Luego, en el parser utilizamos dicha cadena de caracteres para almacenarla en el árbol sintáctico.

Acceso a elementos de un arreglo

Como decisión de diseño optamos por machear el acceso a una posición del arreglo desde el lexer, en lugar de generar producciones en la gramática que lo hicieran. Esto lo hicimos así debido a que, cuando intentamos agregar producciones en la gramática que parsearán el acceso a un elemento del arreglo, se producían múltiples conflictos shift/reduce que no pudimos resolver agregando prioridades.

Por tal razón, controlamos el acceso al arreglo desde el lexer de la siguiente forma:

```
"["[0-9]+"]"          {
                        yylval.string = malloc((strlen(yytext)+1)*sizeof(char));
                        strcpy(yylval.string, yytext);
                        return ARRAY_ACCESS;
                      }
```

En consecuencia, al momento de construir el AST en lenguaje C posteriormente, tuvimos que parsear el entero que indica el índice del elemento en el arreglo para poder buscarlo en el mismo. A su vez, esto nos trae como desventaja que el acceso a los elementos de un arreglo se puede hacer únicamente mediante el uso de enteros positivos, y no negativos como permite ruby. Por lo tanto, se retorna un error en caso de encontrarse accesos como el siguiente: a[-1].

Análisis Sintáctico

Precisamos generar y utilizar un analizador sintáctico que se base en una gramática definida por nosotros. Para ellos utilizamos Bison enlazado con nuestro archivo de Flex para que trabajen en conjunto. Creamos el archivo bison ruby_grammar.y que contiene las producciones de la gramática junto con sus precedencias y estructuras auxiliares. La estructura del archivo es la siguiente:

```
%{
  C declarations
%}
  Bison declarations
%%
  Grammar rules
%%
  Additional C code
```

En la sección de declaraciones en C de la gramática incluimos las librerías de C necesarias para manejar memoria dinámica y cadenas de caracteres, junto con la función “yyerror” que imprime los errores de sintaxis como también las tablas de símbolos general y de clases. La implementación de estas tablas la explicaremos más adelante.

En la sección de declaraciones de Bison definimos los tipos de los símbolos terminales y no terminales, junto con la declaración de los tokens y sus precedencias. Cabe destacar que no incluimos en nuestra gramática algunos de los operadores binarios que se encuentran en la letra del obligatorio. Entre ellos no incluimos los que refieren al macheo de expresiones regulares ni el operador ===.

Producciones de la gramática

Explicaremos en esta sección cómo definimos las reglas gramaticales de nuestro intérprete. Básicamente diseñamos la gramática de tal forma que los archivos de ruby a ser interpretados se componen por una lista de sentencias (pudiendo haber varias sentencias por línea). A su vez, cada sentencia puede estar compuesta por una expresión, una declaración o una línea vacía.

Las declaraciones pueden consistir de una clase, una definición de una función, un iterador (while) o condicionales (if y case). Se agregaron las producciones para cada una de las declaraciones anteriores, pero no se consideró el condicional “case” al implementar el AST ya que no contamos con el tiempo suficiente. Tampoco agregamos el condicional if ternario. Igualmente creemos que habiendo implementado el condicional “if” se puede generar cualquier bifurcación que permitan los condicionales anteriores.

Para el iterador while consideramos que puede tener como condición cualquier tipo de expresión, y en su cuerpo puede contar con una lista de sentencias. Para las clases se considera que las mismas puedan contar opcionalmente con los métodos attr_reader, attr_writter y attr_accessible y con un conjunto opcional de sentencias (la clase puede no contener métodos ni atributos). Se

considera como nombre de una clase una constante (las constantes comienzan con letra mayúscula según el patrón definido en el lexer).

Las definiciones de funciones consideran que los argumentos pasados por parámetros se encuentren o no entre paréntesis, como también se tomó en cuenta que las declaraciones de funciones no cuenten con parámetro alguno.

Las expresiones de la gramática soportan todos los tipos de operadores unarios y binarios, como también asignaciones, llamadas a métodos, y primarios. Los primarios se conforman a su vez por otro conjunto de producciones que definen literales, identificadores, arreglos, accesos a arreglos y el objeto “nil” de ruby.

Por último, los llamados a funciones consideran ser llamados con parámetros y con un bloque opcional (al considerar funciones como el “each”). Cabe destacar que los llamados a funciones con parámetros precisan ser pasados entre paréntesis. Esto lo diseñamos así, ya que de no incorporar paréntesis para el pasaje de parámetros, se producían varios conflictos de tipo shift/reduce.

Árbol Sintáctico

Para construir el árbol sintáctico nos basamos fuertemente en las estructuras utilizadas en el capítulo 3 de [2], ya que nos pareció una forma muy intuitiva y sencilla de generar el mismo. Los archivos que contienen las estructuras de datos, los constructores de los nodos del árbol y su implementación se encuentran en los archivos “structures.h”, “ast.h” y “ast.c” respectivamente. También se cuenta con otros módulos que implementan funciones auxiliares como, por ejemplo, para calcular el largo de una lista.

El AST se compone de nodos, cada uno de los cuales contiene un tipo de nodo (node_type). Distintos tipos de nodos contienen distintos campos, pero se considera el nodo más abstracto “ast_node” como el inicial. Los distintos tipos de nodos tienen sus correspondientes constructores, y cada uno de los mismos es llamado en el código de acción de la producción gramatical que corresponda. Por ejemplo, al ser parseado “1+1” se genera un nodo (ast_node) en el AST de tipo OP_PLUS que contiene un link al operador izquierdo y otro link al operador derecho.

En el parser, se tuvo que hacer uso del constructor %union para declarar los tipos a ser utilizados por los símbolos (terminales y no terminales) de la gramática. En este constructor incluimos los tipos necesarios para construir los nodos del árbol. Una vez que el union está definido, le decimos a Bison el tipo que tiene cada símbolo poniendo el nombre adecuado de cada union entre “< >” a la izquierda de su declaración.

De esta forma generamos un nodo para cada expresión gramatical (que contenga elementos que valga la pena ser guardados) y luego recorremos el AST para evaluarlo como explicamos en la siguiente sección.

Análisis Semántico

Dado que la declaración de variables en Ruby no es tipada, nuestro análisis semántico se realiza al mismo tiempo que se va interpretando el código. Contamos con una función a nivel de C que evalúa un cada del árbol. Aplicamos esa función al nodo raíz de nuestro AST, logrando que recursivamente se ejecute nuestro código de entrada nodo a nodo.

Si bien se recomienda generar representación intermedia, tomamos la decisión de no generarla dado que fue posible evaluar el AST directamente y no contamos con el tiempo para integrarlo. Aún así somos conscientes de sus ventajas, tales como el desacoplamiento que provee entre el Frontend y el Backend.

Aprovechado la información de las estructuras que generamos, en tiempo de ejecución se puede determinar de qué tipo es lo que se está evaluando. Por lo tanto, al momento de ejecutar, si una operación entre dos nodos no es soportada, se devuelve el error correspondiente, pero todo esto se realiza cuando el programa ya está ejecutando.

A modo de ejemplo se presenta el siguiente código de la función que evalúa el árbol para retornar un resultado. En este ejemplo, se distingue cómo se ejecutaría un operador de “mayor que” (greater than), comparando según el tipo de los nodos en cuestión.

```
case N_OP_CMP_GT : {
                    struct ast* left  = eval_ast(node->left);
                    struct ast* right = eval_ast(node->right);

                    // int > int || int > double || double > int || double > double
                    if ((left->node_type == N_INTEGER  || left->node_type == N_DOUBLE) &&
                       (right->node_type == N_INTEGER || right->node_type == N_DOUBLE)) {
                        int value = (double_value(left) > double_value(right)) ? 1 : 0;
                        return new_bool_node(value);
                    }
                   ...
                   ...  /* Distinto soporte de tipos */
                   ...
                   else{
                     type_error(left->node_type, right->node_type);
                   }
```

Ambiente de Ejecución

Mientras ejecutamos la evaluación de nuestro AST, precisamos de estructuras en memoria que guardan y llevanregistro de variables, funciones, clases. Nuestro ambiente de ejecución consta de tres estructuras principales en memoria.

- AST
- Tabla de Símbolos
- Tabla de Clases

AST

Ya fue descripto previamente.

Tabla de Símbolos

Definimos en las declaraciones C del archivo 'ruby_grammar.y' una variable que la utilizaremos como variable global llamada 'sym_table' de tipo 'struct scope*'. Este tipo funciona simplemente como una lista encadenada de scopes (simbolizando un stack) y cada uno de ellos mantiene una lista de símbolos.

En cada lista de símbolos guardamos variables y funciones que hayan sido asignadas o declaradas respectivamente. Contamos con una estructura llamada 'sym' que abarca los dos casos mencionados.

A lo largo del código nos encontramos con sentencias que requieren trabajar sobre distintos scopes, como por ejemplo llamadas a métodos y bloques. Por lo tanto contamos con funciones que nos permites hacer push y pop a nuestro stack de scopes.

```
extern struct scope* sym_table;

struct sym {
  char* name;              /* name of symbol */
  int sym_type;            /* type of symbol: SYM_FUNC || SYM_VAR */
  struct ast* ast;         /* value of symbol */
  struct list_node* args;  /* function arguments */
  struct sym* next;        /* next symbol */
};

struct scope {
  struct sym* sym_list;
  struct scope* next;
};
```

Es importante destacar que al llamar a variables, sólo estarán disponibles si pertenecen a los símbolos al último scope pusheado (scope actual de ejecución).

Tabla de Clases

A medida que se declaran clases, debemos mantener registro de sus declaraciones. Para ello definimos una variable que la utilizaremos como variable global llamada 'class_table' de tipo 'struct class*'.

Cada clase mantiene su propia tabla de símbolos, donde se guardarán sus atributos y métodos asociados. Luego cuando se instancie un objeto y tenga que ir a buscar sus métodos, podrá referenciar y consultar los datos de su clase.

```
extern struct class* class_table;

struct class {
  char* name;
  int sym_type;
  struct sym* sym_list; // SYM_CLASS_VAR, SYM_INST_VAR, SYM_FUNC
  struct class* next;
};
```

Compilación del código y casos de prueba

Para compilar los archivos de nuestro intérprete optamos inicialmente por crear un archivo makefile. Sin embargo, no pudimos escribirlo en forma correcta, ya que la compilación nunca resultaba exitosa. Por tal razón optamos por generar dos archivos scripts: “clean.sh” y “compile.sh” que se comportan de la misma forma que hacer un “make clean” y un “make” respectivamente. También generamos posteriormente un archivo de nombre “all.sh” que ejecuta los scripts anteriores en forma consecutiva, con el único motivo de tipear menos.

Para compilar el archivo “ruby_grammar.y” agregamos la bandera “-v”, que genera un archivo de nombre “ruby_grammar.output”. Este último contiene todos los estados de la máquina de estados generada por Bison para la gramática, junto con los conflictos de tipo shift/reduce y reduce/reduce presentes en la gramática (en caso de existir alguno). Cabe destacar que este archivo nos ayudo en gran forma al momento de tener que resolver los conflictos que surgían en la gramática.

Para testear que el comportamiento del interprete fuera el esperado, se realizaron varias *suites* de tests. Los mismos se encuentran dentro del directorio “tests/” y se separan en unitarios y de integración. Mientras que los unitarios fueron generados con el objetivo de validar operaciones atómicas, los tests de integración buscaron hacer uso de los diferentes componentes del lenguaje.

Los tests pueden ser considerados como los casos de prueba del obligatorio y para correrlos se generó un script “*run_tests.sh*” en el root de la carpeta entregada.

El script *run_tests.sh* genera una salida que verifica si los casos de prueba fueron correctos o no. La misma consiste en una cadena de puntos, donde cada uno representa si la ejecución de un determinado archivo de test fue exitosa o no, siendo un punto verde la representación de un caso exitoso y un punto rojo el contrario.

Para correr la suite entera de casos de prueba se debe seguir los siguientes simples pasos. En el root, ejecutar:

```
>sh all.sh

>sh run_tests.sh
```

En un caso exitoso, esto debería retornar una lista vertical de puntos verdes junto con la cantidad de tests que fueron corridos (contamos con un total de 71 casos de prueba).

Bibliografía

[1] Teórico del curso de la materia, http://www.fing.edu.uy/inco/cursos/compil/teorico.html

[2] Flex & Bison, John Levine, editorial O'Reilly

[3] APIdock, http://apidock.com/

[4] Documentación online de ruby, http://rubydoc.info/

[5] Construcción de compiladores usando Flex y Bison, http://www.fing.edu.uy/inco/cursos/compil/material/compiler-construction-flex-bison.pdf

[6] Ruby in twenty minutes, http://www.ruby-lang.org/en/documentation/quickstart/

[7] Sintaxis BNF del lenguaje ruby, http://www.cse.buffalo.edu/~regan/cse305/RubyBNF.pdf

[8] Writing your own toy compiler using Flex, Bison and LLVM, http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

[9] Using Flex and Bison, http://www.mactech.com/articles/mactech/Vol.16/16.07/UsingFlexandBison/index.html

[10] Compiler construction using Flex and Bison, http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf